Java Art Chapter 0.5. Automatic Wallpapering

This chapter *isn't* about home decoration – the measuring and cutting of strips of wallpaper, the slapping on of paste, the haphazard application to walls. No, I'll be talking about changing the image used as a background (desktop) on your computer screen.

Changing the desktop wallpaper highlights one of Java's strengths – it's platform independence, which paradoxically becomes a weakness when writing applications that interact with the OS. However, as with many criticisms of Java, Java/OS integration *was* a problem in the early days of the language, but a large number of solutions have appeared in recent years. I'll be utilizing JNA, which lets Java dynamically access native OS libraries (https://jna.dev.java.net/). Don't confuse JNA with JNI, the Java Native Interface (http://java.sun.com/javase/6/docs/technotes/guides/jni/), a much older technology. JNI has the same aims as JNA, but is more difficult to master because of its use of C/C++ stub functions. JNA allows a programmer to code entirely in Java, employing interfaces to describe functions and structures in the native library.

I'll be focusing on Windows XP (and Windows 7), but JNA works on a range of platforms, including Linux, OSX, and Solaris.

I want my wallpaper application to change the desktop image without the user's direct intervention; it should occur whenever the machine is first switched on. This brings up another tricky aspect of Java/OS integration – how to get the OS to periodically invoke Java. My answer is to use OS scripting (i.e. a batch file), combined with OS task scheduling (i.e. a cron job). The scripting language acts as a simple interface between the OS and Java.

Application Overview

The application, called GoogleWallpaper, is illustrated in Figure 1.

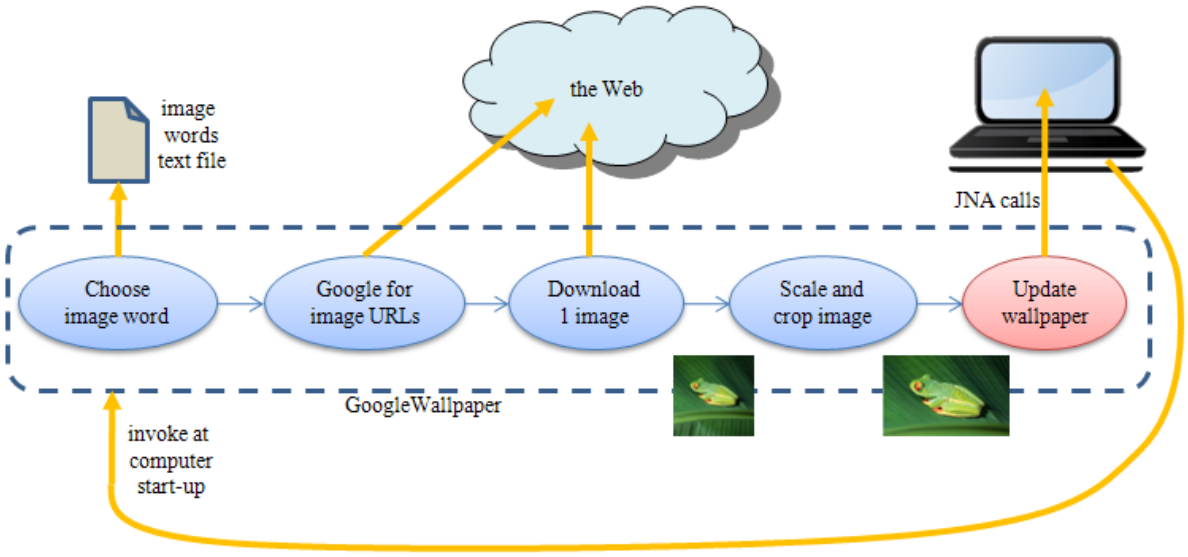

Figure 1. The Main Stages of the GoogleWallpaper Application.

When the machine is switched on, a batch file starts the GoogleWallpaper Java application. It reads in a text file of words, such as:

```
nature
forest
mountain
river
ocean
wilderness
landscape
fields
pasture
```

GoogleWallpaper randomly selects one of the words, and queries Google's image search to obtain URLs of images that match the word. Another Web access (the third oval in Figure 1) downloads one of the images, scaling and cropping it to match the computer's screen dimensions. Updating the desktop image is the last step, and the only one requiring OS-specific functionality. On Windows, this involves changing its registry in three places, and requesting a desktop refresh.

Invoking Java

GoogleWallpaper is started by the ChangeWall.bat batch file:

```
@echo off
echo Executing GoogleWallpaper...
cd /d %~dp0
java -cp "json.jar;jna.jar;platform.jar;." GoogleWallpaper words.txt
```


The java.exe call includes classpaths to JSON and JNA jars, which are located in the same directory as GoogleWallpaper.java, and the name of the image words text file. The call to cd is also important:

```
cd /d %~dp0
```

The "/d" option lets the current drive be changed in addition to the current directory. The "%~dp0" argument applies the 'd' and 'p' batch parameter modifiers to the batch variable %0. %0 returns the batch file's name, 'd' gets its drive, and 'p' its path. These are used by cd to specify a switch to the batch file's own directory. This may seem a bit pointless, since a batch file is normally called from its own directory, like so:

```
> ChangeWall.bat
```

However, I'm going to utilize ChangeWall.bat as an OS scheduling task, which will start in Window's system directory. The "`cd /d %~dp0`" changes the current directory to be the batch's so that java.exe can find the JAR files, the GoogleWallpaper class, and words.txt.

Batch parameters, such as 'd' and 'p', are explained in the XP documentation at http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/percent.mspx?mfr=true.

Scheduling the Batch Script

The easiest way of scheduling a program on Linux is with crontab. Windows offers similar tools: schtasks, at, and the GUI Task Scheduler. Tasks can be scheduled to run one time only, on a minute-by-minute basis, at a specific interval (such as hourly, daily, weekly, or monthly), at system startup, at logon, or whenever the system is idle.

Schtasks has several subcommands, including:

`schtasks /create`	Used to create scheduled tasks.
`schtasks /change`	Used to change the properties of existing tasks.
`schtasks /run`	Used to start a scheduled task immediately.

For example, ChangeWall.bat can be invoked at system start-up with:

```
schtasks /create /tn "Change Wallpaper"
                 /tr c:\scripts\ChangeWall.bat /sc onstart
```

The task name is "Change Wallpaper", and the call assumes that ChangeWall.bat is in c:\scripts\.

Surprisingly, the trickiest aspect of scheduling a script is to make it's invocation invisible to the user. By default, the calling of a batch file causes an ugly looking command window to appear, often flickering into life for only for a second or so. However, it's easy to run the file inside a minimized window by creating a shortcut to it. The shortcut's properties must be changed so that it's "run" field specifies 'run minimized'. The result is that

the batch file's execution will be much less visible, appearing only as a minimized icon. To make the execution totally invisibly is more work, involving VBS scripting with wscript.exe.

I avoided all of these hassles by installing a freeware Windows crontab utility called Z-Cron (http://www.z-cron.com/). This offers a simpler GUI interface than Window's "Scheduled Tasks", and allows a batch file to be run minimized or invisibly. Z-Cron's setup screen for ChangeWall.bat is shown in Figure 2.

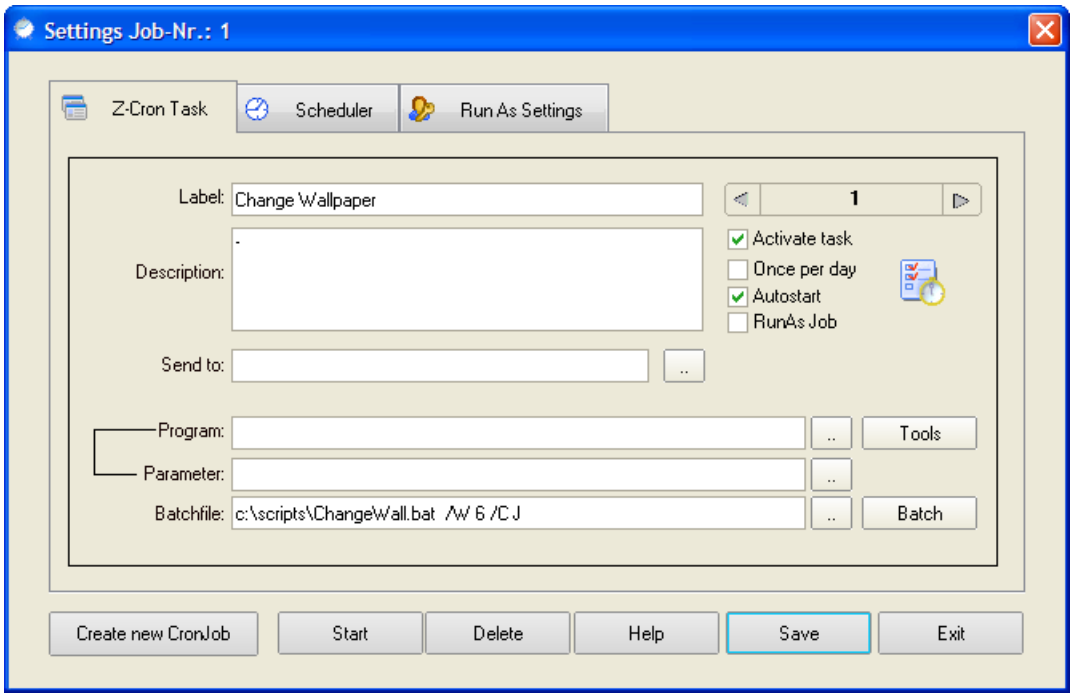

Figure 2. The Z-Cron Setup for ChangeWall.bat

Starting GoogleWallPaper

Figure 1 shows the five main parts of the GoogleWallPaper application (the ovals inside the dashed box), which are also evident in its main() method:

```
// globals
private static final String WALL_FNM = "wallpaper.bmp";
                    // the name of the wallpaper file

private static Random rand;
                    // for selecting a wallpaper at random

private static double screenWidth, screenHeight;
                    // for resizing the wallpaper



public static void main(String args[])
{
  if (args.length == 0) {
    System.err.println("Supply a search words filename");
    System.exit(1);
```



```
  }

  Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
  screenWidth = dim.getWidth();
  screenHeight = dim.getHeight();

  rand = new Random();

  String searchWord = selectSearchWord(args[0]);
  JSONObject json = imageSearch(searchWord);
                              // get search results for the word
  BufferedImage im = selectImage(json);
              // select a result and download its image
  if (im != null) {
    BufferedImage scaleIm = scaleImage(im);     // scale
    BufferedImage cropIm = cropImage(scaleIm);  // crop

    saveBMP(WALL_FNM, cropIm);    // save image in WALL_FNM
    installWallpaper(WALL_FNM);
                           // make WALL_FNM the new desktop wallpaper
  }
}  // end of main()
```

The image retrieved by GoogleWallpaper is saved as a BMP file since XP's desktop expects that format (Windows 7 also accepts JPEGs).

Using Google Image Search

A image word or phrase is selected at random from a list read in from a text file. This phrase is used to query Google's image search, getting back a list of image URLs in JSON format.

The query uses Google's AJAX Search API and its REST-based interface, which is documented at:
http://code.google.com/apis/ajaxsearch/documentation/reference.html#_intro_fonje.

The image search URL is http://ajax.googleapis.com/ajax/services/search/images, followed by a mix of URL arguments to direct the search. The construction of the URL, and the retrieval of the JSON response is handled by the imageSearch() method inside GoogleWallpaper (the second oval in Figure 1):

```
private static JSONObject imageSearch(String phrase)
{
  System.out.println("Searching Google Image for \"" +
                                          phrase + "\"");
  try {
    // Convert spaces to +, etc. to make a valid URL
    String uePhrase = URLEncoder.encode("\"" + phrase + "\"",
                                                 "UTF-8");
    String urlStr =
"http://ajax.googleapis.com/ajax/services/search/images?v=1.0" +
                  "&q=" + uePhrase +
                  "&rsz=large" +      // at most 8 results
                  "&imgtype=photo" +  // want photos
                  "&imgc=color" +     // in colour
                  "&safe=images" +    // Use moderate filtering
```

```
                  "&imgsz=l" ;        // request large images

    String jsonStr = WebUtils.webGetString(urlStr);
                                // get response as a JSON string
    return new JSONObject(jsonStr);
  }
  catch (Exception e)
  { System.out.println(e);
    System.exit(1);
  }
  return null;
}  // end of imageSearch()
```

The URL arguments (the name/value pairs following the '?') are "v" for the search engine version, and "q" for the URL-encoded query. The image phrase is converted to URL-encoded form using URLEncoder.encode(), which replaces spaces and special characters by UTF-8 characters.

WebUtils.webGetString() transmits the query to the service and returns the reply as a string. WebUtils is my own class, holding a range of useful methods for Web querying, DOM parsing, and image downloading. I describe it at length in Chapter 33, “Using Web Service APIs” (http://fivedots.coe.psu.ac.th/~ad/jg/ch33/).

The string returned by Google will usually be lengthy, consisting of multiple JSON structures (name/value pairs and lists). To be more easily readable, the string is parsed into a JSON data structure before being returned to main().

Selecting an Image

Google's JSON result format is explained at http://code.google.com/apis/ajaxsearch/documentation/reference.html#_restUrlBase. The structure is essentially the following:

```
{
  "responseData" : {
    "cursor" : {. . .}    // the useful stuff is in here
    "results" : [. . .],  // and here
  },
  "responseDetails" : null,
  "responseStatus" : 200
}
```

The "`results`" list will contain at most eight matches (a limitation imposed by Google), but the actual total number of matches is stored in "estimatedResultCount" inside "`cursor`":

```
    "cursor": {
      "currentPageIndex": 0,
      "estimatedResultCount": "826000",
          :  // more key/value pairs
    }
```

The tuples and lists can be examined using the org.json get methods getJSONObject(), getJSONArray(), and getString(), which come from the json.jar package added to GoogleWallpaper's classpath.

selectImage() reports the total number of matches (by accessing the "estimatedResultCount" field), prints the contents headers and URLs of the matches by calling showResults(), and returns a randomly selected image downloaded from one of the URLs via tryDownloadingImage():

```
private static BufferedImage selectImage(JSONObject json)
{
  try {
    System.out.println("\nTotal no. of possible results: " +
                    json.getJSONObject("responseData")
                        .getJSONObject("cursor")
                        .getString("estimatedResultCount") + "\n");

    // list search results and download one of their images
    JSONArray jaResults =
      json.getJSONObject("responseData").getJSONArray("results");
    showResults(jaResults);
    if (jaResults.length() > 0)
      return tryDownloadingImage(jaResults);
  }
  catch (JSONException e)
  { System.out.println(e);
    System.exit(1);
  }

  return null;
} // end of selectImage()
```

Each matching result consists of many key/value pairs, including the image's title, dimensions, and its URL. For example:

```
    "results": [
      {
        "contentNoFormatting": "Anapsos] Rain Forest ferns",
        "height": "768",
        "title": "Polypodium Leucotomos << Resource Site",
        "url": "http://gentlehugs.files.wordpress.com/2009/05/
                                              rain_forest_tropic.jpg",
        "width": "1024"
            :     // more key/value pairs
      },
      :  // more results tuples {. . .}
    ]
```

showResults() cycles through each result, printing its contents title and URL:

```
private static void showResults(JSONArray jaResults)
                                              throws JSONException
{ for (int i = 0; i < jaResults.length(); i++) {
    System.out.print((i+1) + ". ");
    JSONObject j = jaResults.getJSONObject(i);
    String content = j.getString("contentNoFormatting");
    String cleanContent = content.replaceAll("[^a-zA-Z0-9]", " ").
                                  replaceAll("\\s+", " ").
                                  trim();
     // replace non-alphanumerics with spaces; remove multiple spaces
    System.out.println("Content: " + cleanContent);
    System.out.println("       URL: "+ j.getString("url") + "\n");
```



```
  }
}  // end of showResults()
```

Some typical output is:

```
Total no. of possible results: 8290

1. Content: Gerhard Richter Seascape
       URL: http://palad1n.com/images/seascape.jpg

2. Content: Hawaiian Seascape on Golden
       URL: http://wallpapers-
diq.com/wallpapers/42/Hawaiian_Seascape_on_Golden_
Sunset,_Hawaii.jpg

3. Content: Fermin Seascape San
       URL:
http://www.zastavki.com/pictures/1024x768/2008/World_USA_Point_Fermi
n_Seascape___San_Pedro___California___USA_008938_.jpg
                 :
```

tryDownloadingImage() download an image from a URL chosen at random from the results list. This is complicated by the possibility that the URL is unavailable.

```
// global
private static final int MAX_TRIES = 5;
               // max number of times to try download an image


private static BufferedImage tryDownloadingImage(JSONArray jaResults)
                                          throws JSONException
{ BufferedImage im = null;
  for(int i=0; i < MAX_TRIES; i++) {
    int idx = rand.nextInt(jaResults.length());
                          // select an image index at random
    System.out.println("Randomly selected no. " + (idx+1));
    String imUrlStr = jaResults.getJSONObject(idx).getString("url");
                            // get its URL
    im = getURLImage(imUrlStr);   // download the URL (maybe)
    if (im != null)
      return im;
  }

  // should not get here unless there's a problem
  System.out.println("No suitable image found");
  return im;
}  // end of tryDownloadingImage
```

If a URL is unavailable, tryDownloadingImage() makes another random choice, and there's a slim chance that it will try to download the same URL again. However, the code has up to MAX_TRIES attempts to find a good link, so I haven't complicated the algorithm by forcing each choice to be unique.

getURLImage() uses Java's URL and ImageIO classes to return the URL's data as a BufferedImage:

```
private static BufferedImage getURLImage(String urlStr)
{
  System.out.println("Downloading image at:\n\t" + urlStr);
  BufferedImage image = null;
  try {
    image = ImageIO.read( new URL(urlStr) );
  }
  catch (IOException e)
  {  System.out.println("Problem downloading");  }

  return image;
}  // end of getURLImage()
```

Knocking the Image into Shape

The image returned by selectImage() will almost certainly *not* be the correct size and shape for the computer's screen, so GoogleWallpaper carries out a two-stage shape adjustment – scaling followed by cropping (see Figure 3).

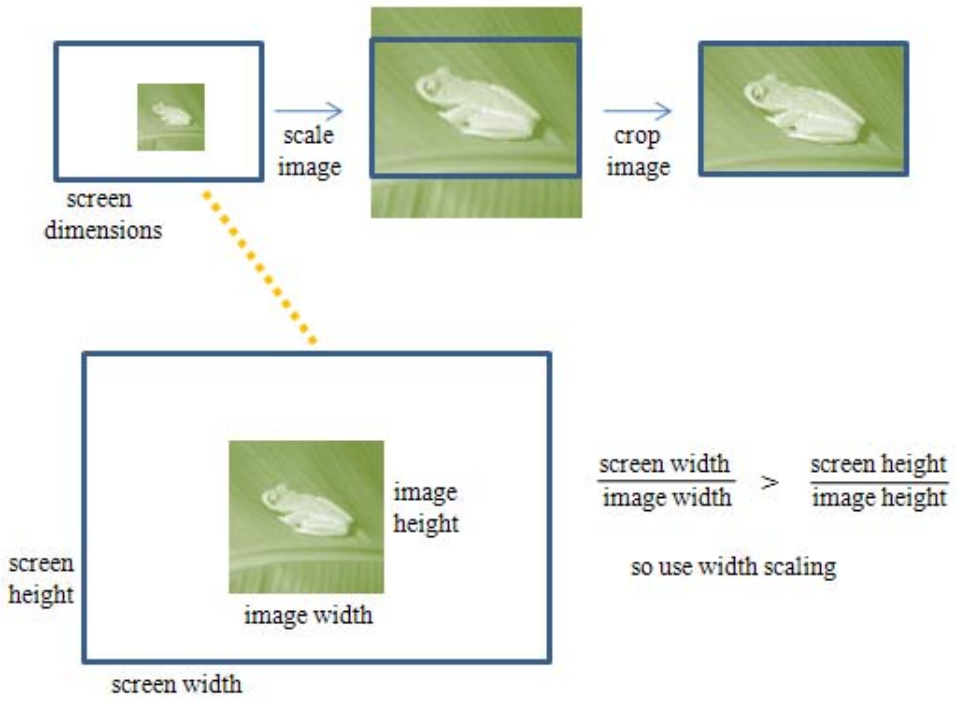

Figure 3. Scaling and Cropping the Image for the Screen.

The same scaling is applied to the image's width and height, to preserve its proportions. The scaling value is arrived at by calculating two scaling ratios for the screen/image widths and heights. In Figure 3, the screen/image width ratio is about 3, while the screen/image height ratio is around 2. The larger of the two ratios is chosen as the scaling factor. This is implemented by the scaleImage() method:

```
// globals
private static double screenWidth, screenHeight;
```



```
private static BufferedImage scaleImage(BufferedImage im)
{
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();

  // calculate screen-dimension/image-dimension for width and height
  double widthRatio = screenWidth/(double)imWidth;
  double heightRatio = screenHeight/(double)imHeight;

  double scale = (widthRatio > heightRatio) ?
                                widthRatio : heightRatio;
       // scale is the largest screen-dimension/image-dimension
  // calculate new image dimensions
  int scWidth = (int)(imWidth*scale);
  int scHeight = (int)(imHeight*scale);

  // resize the image
  BufferedImage scaledImage = new BufferedImage(scWidth, scHeight,
                                          BufferedImage.TYPE_INT_RGB);
  Graphics2D g2d = scaledImage.createGraphics();
  AffineTransform at =
                AffineTransform.getScaleInstance(scale, scale);
  g2d.setRenderingHint( RenderingHints.KEY_INTERPOLATION,
                        RenderingHints.VALUE_INTERPOLATION_BICUBIC);
  g2d.drawImage(im, at, null);
  g2d.dispose();

  return scaledImage;
}  // end of scaleImage()
```

scaleImage() calculates the screen/image width and height ratios, and stores the larger value in the scale variable. This is used to calculate the dimensions of a new BufferedImage, and the downloaded image is rendered into it; interpolation is used to smooth the resizing.

The result is an image which is the same size as the screen in one dimension (e.g. across its width in Figure 3), and bigger than the screen along the other (the height in Figure 3).

Cropping is now a matter of finding out which one of the two dimensions (width or height) is bigger than the screen, and removing the excess. The editing is done to both ends of the dimension (e.g. to the top *and* bottom of the excess height in Figure 3) to ensure that the image's center stays in the center of the screen. cropImage() implements this technique:

```
private static BufferedImage cropImage(BufferedImage scIm)
{
  int imWidth = scIm.getWidth();
  int imHeight = scIm.getHeight();

  BufferedImage croppedImage;
  if (imWidth > screenWidth) {  // image width > screen width
    // System.out.println("Cropping the width");
    croppedImage = new BufferedImage((int)screenWidth, imHeight,
                              BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = croppedImage.createGraphics();
```

```
    g2d.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                          RenderingHints.VALUE_ANTIALIAS_ON);
    int x = ((int)screenWidth - imWidth)/2;
             // crop so image center remains in the center
    g2d.drawImage(scIm, x, 0, null);
    g2d.dispose();
  }
  else if (imHeight > screenHeight) { //image height > screen height
    // System.out.println("Cropping the height");
    croppedImage = new BufferedImage(imWidth, (int)screenHeight,
                                     BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = croppedImage.createGraphics();
    g2d.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                          RenderingHints.VALUE_ANTIALIAS_ON);
    int y = ((int)screenHeight - imHeight)/2;
             // crop so image center remains in the center
    g2d.drawImage(scIm, 0, y, null);
    g2d.dispose();
  }
  else   // do nothing
    croppedImage = scIm;

  return croppedImage;
}  // end of cropImage()
```

Another new BufferedImage object is created, and the current image drawn into it using carefully chosen top-left (x, y) coordinates to make sure the image remains centered.

The functionality in scaleImage() and cropImage() could be combined, thereby reducing the amount of image copying, but the method separation makes the code easier to understand.

Updating the Desktop Wallpaper

Wallpaper installation requires three changes to the Windows registry, and a desktop refresh. The basic idea (using Visual C# and VB) is explained in "Setting Wallpaper" by Sean Campbell at http://blogs.msdn.com/coding4fun/archive/2006/10/31/912569.aspx. Another useful example, written in C and Japanese, can be found at http://www9.plala.or.jp/NAT/program/sw/c/changeWallPaper.c.

Modifying the Registry

The three relevant Windows registry keys are below the HKEY_CURRENT_USER\Control Panel\Desktop branch:

Wallpaper	Its value should be set to the path to bitmap file.
WallpaperStyle	Its value should be set to 0 so the image will not be stretched.
TileWallpaper	Its value should be 0 to indicate no image tiling.

When WallpaperStyle and TileWallpaper are both 0, the wallpaper will be displayed in the center of the screen.

HKEY_CURRENT_USER\Control Panel\Desktop is home to a large number of important desktop-related keys, which are detailed at http://www.virtualplastic.net/html/desk_reg.html.

There's numerous ways of manipulating the Windows registry via Java. For example, by calling the Windows reg command via Runtime.getRuntime().exec() (e.g. see http://www.rgagnon.com/javadetails/java-0480.html and http://stackoverflow.com/questions/62289/read-write-to-windows-registry-using-java), or by using libraries such as jRegistryKey (http://sourceforge.net/projects/jregistrykey/) and JNIRegistry (http://www.trustice.com/java/jnireg/). I decided to use JNA (Java Native Access; https://jna.dev.java.net/) because it not only offers a simplified registry interface, but can also access other native shared libraries (i.e. any DLL in Windows), which I need for refreshing the desktop after modifying the registry.

Windows' registry API is quite extensive, and a list of its functions can be found at the MSDN library page http://msdn.microsoft.com/en-us/library/ms724875(v=VS.85).aspx. Essentially a key must be opened, its value set, with RegSetKeyValue(), and then the key closed. The DLL holding these functions, Advapi32.dll, can be easily accessed through JNA, but JNA also offers an Advapi32Util class which simplifies many operations.

JNA comes as a single jna.jar file, but the Windows-specific Advapi32Util class also requires the platform.jar file (both available at https://jna.dev.java.net/). I placed these in the same directory as GoogleWallpaper, and added them to its classpath:

```
java -cp "json.jar;jna.jar;platform.jar;." GoogleWallpaper words.txt
```

The Advapi32Util class is located in the Platform-specific package com.sun.jna.platform.win32, which is documented at https://jna.dev.java.net/javadoc/platform/com/sun/jna/platform/win32/package-summary.html. Advapi32Util contains about 20 registry-related methods.

GoogleWallpaper's installWallpaper() method uses the registry set methods:

```
private static void installWallpaper(String fnm)
{
  try {
    String fullFnm = new File(".").getCanonicalPath() + "\\" + fnm;

    Advapi32Util.registrySetStringValue(WinReg.HKEY_CURRENT_USER,
              "Control Panel\\Desktop",
                  "Wallpaper", fullFnm);
    Advapi32Util.registrySetIntValue(WinReg.HKEY_CURRENT_USER,
              "Control Panel\\Desktop",
                  "WallpaperStyle", 0);    // no stretching
    Advapi32Util.registrySetIntValue(WinReg.HKEY_CURRENT_USER,
              "Control Panel\\Desktop",
                   "TileWallpaper", 0);    // no tiling

    // Refresh the desktop: explained below . . .
    //        :
  }
  catch(IOException e)
```

```
  {  System.out.println("Could not find directory path");  }
}  // end of installWallpaper()
```

The Advapi32Util methods hide registry key opening and closing, and support string and integer value arguments.

Refreshing the Desktop

After the registry changes, it's still necessary to refresh the desktop by updating its configuration information. This can be achieved by calling SystemParametersInfo() from the User32 DLL. SystemParametersInfo() is a rather complex function since it can be used to modify a wide range of different features, including the desktop, icons, menus, power settings, the screen saver, time-outs, and GUI effects. It is documented in the MSDN library at http://msdn.microsoft.com/en-us/library/ms724947(VS.85).aspx.

The relevant parameter for refreshing the wallpaper is SPI_SETDESKWALLPAPER. In that use-case, SystemParametersInfo() also requires the location of the bitmap file and profile bit values SPIF_UPDATEINIFILE and SPIF_SENDWININICHANGE (or SPIF_SENDCHANGE). SPIF_UPDATEINIFILE causes the user profile to be updated, and SPIF_SENDWININICHANGE broadcasts a WM_SETTINGCHANGE message to all the top-level windows to notify them of that change.

The Microsoft article “How to Use SystemParametersInfo API for Control Panel Settings” (http://support.microsoft.com/kb/97142) kindly supplies an example of this version of SystemParametersInfo(). In Visual Basic, the function definition is:

```
Const SPI_SETDESKWALLPAPER = 20
Const SPIF_UPDATEINIFILE = &H1
Const SPIF_SENDWININICHANGE = &H2

Declare Function SystemParametersInfo Lib "User" (
           ByVal uAction As Integer,
           ByVal uparam As Integer,
           ByVal lpvParam As String,
           ByVal fuWinIni As Integer ) As Integer
```

This doesn’t quite match the SystemParametersInfo() documentation at http://msdn.microsoft.com/en-us/library/ms724947(VS.85).aspx which specifies the library as User32.dll and the return type as BOOL (i.e. a boolean).

The function is called like so in VB:

```
filenm$ = "C:\Windows\rivets.bmp"
result% = SystemParametersInfo(
                  SPI_SETDESKWALLPAPER, 0&, filenm$,
                  SPIF_UPDATEINIFILE Or SPIF_SENDWININICHANGE )
```

The second argument of SystemParametersInfo() isn't needed when adjusting the wallpaper, so is set to zero.

JNA includes many pre-defined Java bindings for User32 functions in its Platform-specific package com.sun.jna.platform.win32, as documented at

https://jna.dev.java.net/javadoc/platform/com/sun/jna/platform/win32/package-summary.html. The bad news is that SystemParametersInfo() isn't one of them.

I need to create an interface based on JNA's StdCallLibrary class which defines the User32 DLL and the function(s) that I want. There's no need for any C/C++ stub code, and the mapping from Win32 data structures to Java types is quite intuitive.

My MyUser32 interface defines User32.SystemParametersInfo():

```
private interface MyUser32 extends StdCallLibrary
{
   MyUser32 INSTANCE =
       (MyUser32) Native.loadLibrary("user32", MyUser32.class);

   boolean SystemParametersInfoA(int uiAction, int uiParam,
                                 String fnm, int fWinIni);
}  // end of MyUser32 interface
```

MyUser32 loads the User32.dll, and utilizes the SystemParametersInfoA() function. Note the "A" at the end of the function name, which isn't a typo.

When I first wrote MyUser32, I naturally tried to call SystemParametersInfo() (no 'A'). The result was an UnsatisfiedLinkError raised by JNA at runtime. I quickly turned to "DLL Export Viewer" (http://www.nirsoft.net), a freeware utility that can list all the exported functions from a DLL. The relevant output for User32.dll is shown in Figure 4.

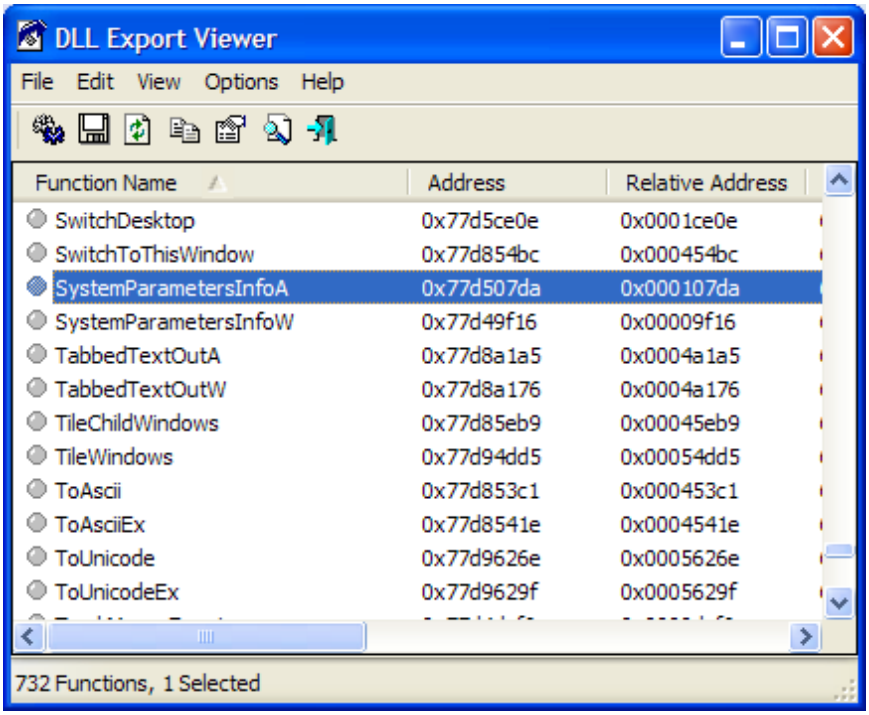

Figure 4. Exported Functions from User32.dll.

There's no SystemParametersInfo() function in User32.dll; instead there's SystemParametersInfoA() and SystemParametersInfoW(), which are ANSI and Unicode string versions of the function. I don't need Unicode support, and so MyUser32 now employs SystemParametersInfoA().

In GoogleWallpaper's installWallpaper(), SystemParametersInfoA() is called in a very similar way to the VB example from above:

```
private static void installWallpaper(String fnm)
{
  try {
    String fullFnm = new File(".").getCanonicalPath() + "\\" + fnm;

    // registry setting code; described above . . .

    // refresh the desktop
    int SPI_SETDESKWALLPAPER = 0x14;   // 20 in decimal
    int SPIF_UPDATEINIFILE = 0x01;
    int SPIF_SENDWININICHANGE = 0x02;

    boolean result = MyUser32.INSTANCE.SystemParametersInfoA(
          SPI_SETDESKWALLPAPER, 0, fullFnm,
          SPIF_UPDATEINIFILE | SPIF_SENDWININICHANGE );
    System.out.println("Refresh desktop result: " + result);
  }
  catch(IOException e)
  {  System.out.println("Could not find directory path");  }
}  // end of installWallpaper()
```

If the desktop is updated successfully then the result boolean will be set to true. More obviously, the desktop will start displaying a new picture.

More on JNA

The JNA website contains a very readable "Getting Started" guide (https://jna.dev.java.net/), and there are a number of good online articles with plenty of examples:

> "Open source Java projects: Java Native Access" by Jeff Friesen, http://www.javaworld.com/javaworld/jw-02-2008/jw-02-opensourcejava-jna.html
>
> "More JNA Examples' by Jeff Friesen, http://javajeff.mb.ca/cgi-bin/mp.cgi?/java/javase/articles/mjnae. A PDF version is available at http://javajeff.mb.ca/java/javase/ebooks/mjnae/mjnae.pdf
>
> "Protect Your Legacy Code Investment with JNA" by Stephen B. Morris, http://today.java.net/pub/a/today/2009/05/19/protect-your-legacy-code-jna.html
>
> "Simplify Native Code Access with JNA" by Sanjay Dasgupta, http://today.java.net/article/2009/11/11/simplify-native-code-access-jna

Wallpapering Problems

I've tested GoogleWallpaper on several Windows XP and Windows 7 machines, but there's still a chance that it may not work for some users. A good way of determining whether its my code or the OS that's at fault is to try to manually change the desktop wallpaper.

On Windows XP, the wallpaper can be altered by the user right-clicking on the desktop and selecting the Properties menu item, which will bring up the "Display

Properties" window. The Desktop tab should be chosen, and its Browse button used to find a BMP file to act as wallpaper. Unfortunately, there's several things that might go wrong:

1. The Desktop Tab may be missing from the "Display Properties" window. A fix can be found at http://www.theeldergeek.com/desktop_tab_missing_from_display.htm
2. All the options may be disabled on the Desktop tab, making it impossible to choose a wallpaper file. One solution is offered at http://malektips.com/windows_xp_display_desktop_0002.html. This may cause Window's Active Desktop to be enabled, which needs to be turned off (see the next point).
3. Active Desktop is enabled (as shown by a quick look at the desktop), preventing a file being assigned as the wallpaper. Go to the Desktop tab, and click on the "Customize Desktop" button. In the resulting "Desktop Items" window, select the Web tab, and uncheck all its entries.
4. Active Desktop is active, but there's no Web tab to switch it off. To add the tab back to the window, follow the steps at http://www.winxptutor.com/webtab.htm.

On Windows 7, the manual changing of the wallpaper is almost the same as on XP: right-click on the desktop, selecting the Personalize menu item. In the resulting window, click on the "Desktop Background" button. Press the Browse button, and supply either a JPEG or BMP file; hit "Save changes". Several things may go wrong:

1. The number one problem with manual wallpapering in Windows 7 is the Starter edition of the OS, which disables wallpaper adjustment. There's no Personalize menu item, and the \\HKEY_CURRENT_USER\Control Panel\Desktop\Wallpaper key is hard-coded to point to %windir%\web\wallpaper\windows\img0.jpg. Even if that filename is reused to hold a different picture, Windows checks its contents at startup time, and will display a black background if it's been changed.

   The simplest solution is to upgrade to a more fully featured version of Windows. If that isn't possible then there are complicated registry fixes that can get around the restrictions, as detailed at http://www.withinwindows.com/2009/03/31/correction-starter-wallpaper-more-secure-than-i-thought/comment-page-2/#comment-5295 (in particular, the posts by srg84 and Gabe).
2. The manual changing of the wallpaper may seem to be progressing smoothly, but no changes appear on-screen after "Save changes" is pressed. One fix is to use regedit to examine the HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\system branch, and delete its Wallpaper and WallpaperStyle keys.
3. It may only be possible to change the background image to a solid color. This can be remedied by deleting the file %USERPROFILE%\AppData\Roaming\Microsoft\Windows\Themes\TranscodedWallpaper.jpg (%USERPROFILE% is usually c:\users\YourUsername\).